Lab6 - Peticiones AJAX GET

Pepe Calderón, Fernando Sola, Daniel Ayala, Inma Hernández, Margarita Cruz, Carlos Arévalo, David Ruiz

Escuela Técnica Superior de
Ingeniería Informática

Índice

1 Objetivo

El objetivo de esta práctica es introducir la comunicación del front-end implementado, usando HTML+CSS+JS, con el back-end del proyecto, consistente en una serie de endpoints REST. El alumno aprenderá a:

- Usar la librería JavaScript "Axios" para realizar peticiones AJAX de tipo GET a los endpoints del proyecto.
- Encapsular el código encargado de realizar este tipo de peticiones en módulos JavaScript de comunicación con la API del proyecto mediante programación asíncrona usando `async/await`.
- Usar y mostrar los datos obtenidos en las diferentes vistas, gestionando los posibles errores de manera adecuada.

2

Introducción

Hasta ahora, los datos mostrados en nuestra aplicación han sido introducidos a mano, ya sea directamente en el HTML de la página o definiéndolos mediante JavaScript para su posterior renderizado. Sin embargo, en aplicaciones reales estos datos provendrán de un servidor, normalmente a través de una API REST y en formato JSON. Por ello, es muy importante saber acceder a estos datos para interactuar con ellos en el front-end.

Tradicionalmente, este tipo de peticiones a un servidor por parte de un navegador mediante JavaScript reciben el nombre de peticiones AJAX (**A**synchronous **J**avaScript **A**nd **X**ML), ya que se realizan de manera asíncrona para no bloquear la página durante la espera a que el servidor responda. Pese a que XML está en desuso como formato de intercambio de datos en favor de JSON, el acrónimo AJAX se sigue usando para denominar a estas peticiones.

En esta práctica aprenderemos a consultar algunos endpoints GET definidos en nuestro proyecto. Para ello, haremos uso de la librería Axios, ampliamente usada actualmente y que simplifica el proceso de hacer peticiones AJAX. Aprenderemos también a encapsular el código responsable de estas peticiones en módulos especializados, para simplificar el código de las vistas y poder reutilizar las peticiones en tantos sitios como sea necesario.

3

Configurando la conexión a la API

Todo lo relativo a las peticiones a la API del proyecto usando JavaScript se encuentra en la carpeta `web/js/api/`. Observe que en esta carpeta se incluye por defecto un archivo, `common.js`, que contiene definiciones comunes que son de utilidad para cualquier petición que hagamos:

```
const BASE_URL = "/api/v1";

const requestOptions = {
    headers: { Token: sessionManager.getToken() },
};
```

La constante `BASE_URL` hace referencia al prefijo de la URL que tenemos configurado en nuestro proyecto. **Es importante asegurarse de que es correcto**, ya que si no todas las peticiones fallarán al no incluir el prefijo correcto. El objeto `requestOptions` define las opciones y cabeceras que se enviarán en todas las peticiones. En este caso, está relacionado con el envío automático de tokens de sesión, que no son relevantes por el momento.

4

Módulos API

En el proyecto, los módulos JavaScript responsables de hacer peticiones a los diferentes endpoints de la aplicación se encuentran en la carpeta `/js/api/`. Dado que las operaciones que se suelen realizar contra estos endpoints son, en la mayoría de casos, peticiones CRUD, Silence permite generar automáticamente los módulos correspondientes a cada uno de ellos. Mediante el comando `silence createapi`, se crean en esta carpeta de manera automática los módulos correspondientes a los endpoints definidos en nuestro proyecto.

Para cada uno de los recursos almacenados en nuestra base de datos, tendremos un módulo que se encargará de realizar las peticiones correspondientes contra el mismo. El nombre de los módulos autogenerados comienza con guión bajo (_) seguido del nombre del recurso. Esto permite poder crear manualmente módulos adicionales para peticiones más complejas que puedan ser necesarias. Por ejemplo, las operaciones CRUD básicas del endpoint de fotos se encuentran el el archivo autogenerado `/js/api/_photos.js`.

{: .warning }

Info

No modifique los módulos generados automáticamente, ya que los cambios se perderán si se vuelve a ejecutar el comando `silence createapi`. En su lugar, si necesita usar peticiones diferentes a las autogeneradas, cree un nuevo módulo y añada el código necesario para realizar las peticiones correspondientes.

Las peticiones GET se implementan como métodos dentro de un objeto definido en el módulo correspondiente. Por ejemplo, el método autogenerado para consultar todas las fotos es:

```
getAll: async function() {
   let response = await axios.get(`${BASE_URL}/photos`,requestOptions);
   return response.data;
},
```

Observe lo siguiente:

- Todos los métodos de petición deben ser declarados como `async` para que puedan ser usados en una función asíncrona.
- Se realiza la petición usando la librería `axios`, mediante el método `axios.get()`.
- La URL se construye a partir de la URL base común, definida en el archivo `/js/api/common.js`, que debemos importar.
- La llamada a `axios.get()` es asíncrona, por lo que se debe convertir en síncrona para esperar a que el servidor responda. Para ello, se utiliza `await` antes de la llamada, convirtiéndola en bloqueante.
- Una vez el servidor responde, se devuelven los datos de la respuesta, que se encuentran en el atributo `data` del objeto `response`. Axios se encarga de interpretar los datos JSON de la respuesta y convertirlos a objetos y arrays JavaScript.
- El parámetro `requestOptions`, importado también de `common.js`, incluye los datos que se deben enviar en todas las peticiones al servidor, por ejemplo, el token de sesión.

En las siguientes secciones, usaremos estos módulos autogenerados para realizar las peticiones a los endpoints de la aplicación.

5 Renderizando las fotos del servidor

En prácticas anteriores, definíamos manualmente el conjunto de fotos a mostrar en la página principal de la galería, en el archivo `index.js`. Ahora que contamos con módulos JavaScript con los que poder consultar las fotos existentes en la base de datos a través de la API, podemos usarlos para obtener por esa vía el array de fotos a mostrar.

Para ello, debemos realizar dos modificaciones mínimas en el HTML de la vista principal: en primer lugar, incluiremos el `<script>` correspondiente para importar Axios en el `<head>`:

```
<script src="js/libs/axios.min.js"></script>
```

Además, añadiremos un div de errores dentro del contenedor principal:

```
<div id="errors"></div>
```

Con estos cambios realizados, podemos cambiar `index.js` para que utilice el módulo relativo a la API de fotos:

```
"use strict";

import { photosAPI_auto } from "/js/api/_photos.js";
import { galleryRenderer } from "/js/renderers/gallery.js";
import { messageRenderer } from "/js/renderers/messages.js";

async function main() {
    loadAllPhotos();
}
```

```
async function loadAllPhotos() {
    let galleryContainer = document.querySelector("div.container");

    try {
        let photos = await photosAPI_auto.getAll();
        let cardGallery = galleryRenderer.asCardGallery(photos);
        galleryContainer.appendChild(cardGallery);
    } catch (err) {
        messageRenderer.showErrorMessage("Error while loading photos",
err);
    }
}

document.addEventListener("DOMContentLoaded", main);
```

Observe las siguientes consideraciones:

- La función `loadAllPhotos()` es una función asíncrona, dado que realiza una petición a la API. Esto permite que pueda ser ejecutada en segundo plano, sin que la página se bloquee.
- La llamada a `photosAPI_auto.getAll()` es asíncrona, pero debemos esperar a que el servidor responda para poder renderizar las fotos. Por ello, utilizamos `await` antes de la llamada.
- El formato de las fotos es el mismo que el utilizado anteriormente, por lo que podemos pasar la lista de fotos al renderizador correspondiente y mostrar la galería en la página.
- Si ocurre cualquier problema durante la carga de las fotos, se ejecutará el `catch`, capturando la excepción y mostrando un mensaje en el div de errores.
- La función de entrada `main()` también debe ser `async`, ya que en su interior se usan funciones asíncronas.
- La llamada a `loadAllPhotos()` se ejecuta en segundo plano al ser esta función asíncrona, por lo que se podrían seguir ejecutando instrucciones en la función `main()` sin esperar a que el servidor responda la petición.

6

Vista de detalle de foto

En sesiones anteriores, implementamos una vista de detalle de foto que, gracias a un renderizador, muestra una foto concreta. Sin embargo, esta foto está establecida por nosotros, y sería deseable que se adaptara para mostrar cualquier foto posible: por ejemplo, que al pulsar en una foto de la galería principal se mostrara ésta en detalle en `photo_detail.html`.

Para ello, es necesario que la vista de detalle de foto reciba información sobre qué foto debe mostrar (generalmente se usa el ID de la foto). La forma más habitual de lograr esto es mediante parámetros de URL: por ejemplo, se puede acceder a esta vista mediante la URL `(...)/photo_detail.html?photoId=X`. En este caso, el navegador carga la página `photo_detail.html` como es habitual, pero el parámetro `photoId` proporcionado en la URL puede ser leído mediante JavaScript.

Recordemos que el renderizador de fotos permite mostrar una foto como tarjeta, que incluye un enlace a `photo_detail.html`. Podemos editar el **renderizador de fotos** para que el enlace de cada foto en formato tarjeta incluya su `photoId` como parámetro de URL:

```
asCard: function(photo) {
  // ...código anterior...
  let html = `
    <a href="photo_detail.html?photoId=${photo.photoId}">
        <img src="${photo.url}" class="card-img-top">
    </a>
  `;
  // ...resto del código...
}
```

Esto hace que, al pulsar en una foto de la galería, se transmita la información relativa al `photoId` de la foto seleccionada mediante un parámetro de URL. Podemos capturar este parámetro usando JavaScript, en `photo_detail.js`:

```
let urlParams = new URLSearchParams(window.location.search);
let photoId = urlParams.get("photoId");
console.log("The photo ID to load is: " + photoId);
```

El objeto `URLSearchParams` sirve para acceder más fácilmente a los parámetros de URL, que se encuentran en `window.location.search`. Con este objeto, podemos acceder a un parámetro determinado usando `urlParams.get()`. Esto hará que se muestre por consola el ID de la foto que debemos mostrar:

```
The photo ID to load is: 6
```

Teniendo el ID de la foto, podemos hacer una consulta a la API para obtener los datos de la misma, y proporcionárselos al renderizador para que muestre la foto en cuestión, de manera muy similar a la usada para la galería:

```
"use strict";

import { photoRenderer } from "/js/renderers/photos.js";
import { photosAPI_auto } from "/js/api/_photos.js";
import { messageRenderer } from "/js/renderers/messages.js";

// Get the ID of the photo to load from the URL params
let urlParams = new URLSearchParams(window.location.search);
let photoId = urlParams.get("photoId");

async function main() {
    // Check that we have an ID before doing anything else
    if (photoId === null) {
        messageRenderer.showErrorMessage("Please, provide a photoId");
        return;
    }

    loadPhotoDetails();
}

async function loadPhotoDetails() {
    let photoContainer = document.querySelector("#photo-details-column");
    try {
        let photo = await photosAPI_auto.getById(photoId)
        let photoDetails = photoRenderer.asDetails(photo);
        photoContainer.appendChild(photoDetails);
    } catch (err) {
        messageRenderer.showErrorMessage("Error loading photo", err);
    }
}
```

```
document.addEventListener("DOMContentLoaded", main);
```

En este caso, podemos definir `urlParams` y `photoId` fuera de la función `main()` ya que los parámetros de URL están disponibles en todo momento, no sólo cuando la página está completamente cargada, así, podemos usar `photoId` en cualquier lugar del código desarrollado en el archivo.

Finalmente, para que todos los elementos funcionen correctamente, es importante modificar la vista `photo_detail.html` para añadir:

- La etiqueta `<script src="js/libs/axios.min.js"></script>` para importar Axios.
- El contenedor ~para mostrar los posibles errores.

7

Peticiones AJAX a vistas

En ocasiones, para proporcionar una buena experiencia de usuario, es necesario realizar peticiones a varios recursos a la vez. Por ejemplo, a la hora de mostrar las tarjetas de la galería, puede ser buena idea mostrar el nombre de usuario que subió la foto junto con su imagen de perfil. Sin embargo, la tabla `Photos` sólo contiene el `userId` de la foto. En estos casos, es conveniente crear una vista que contenga los campos adicionales que deseamos mostrar, y realiar la petición GET a la vista en cuestión.

En la plantilla de proyecto utilizada, la base de datos contiene una vista `PhotosWithUsers`, que devuelve todas las filas de la tabla `Photos` y adicionalmente, para cada una de ellas, el nombre del usuario que subió la foto y la URL de su avatar.

Los módulos JS autogenerados para consultar la API incluyen también peticiones de tipo GET para las vistas de la BD. Concretamente, las peticiones para la vista anterior se encuetran en el archivo autogenerado `/js/api/_photoswithusers.js`.

Para implementar esta modificación, debemos cambiar el módulo usado para hacer la petición en `index.js`, usando el de esta vista en lugar del de las fotos:

```
import { photoswithusersAPI_auto } from "/js/api/_photoswithusers.js";

(...)

async function loadAllPhotos() {
    let galleryContainer = document.querySelector("div.container");

    try {
```

```
        let photos = await photoswithusersAPI_auto.getAll();
    (...)
```

A continuación, debemos acomodar esta nueva información en el renderizador de fotos, para que se muestre en la vista:

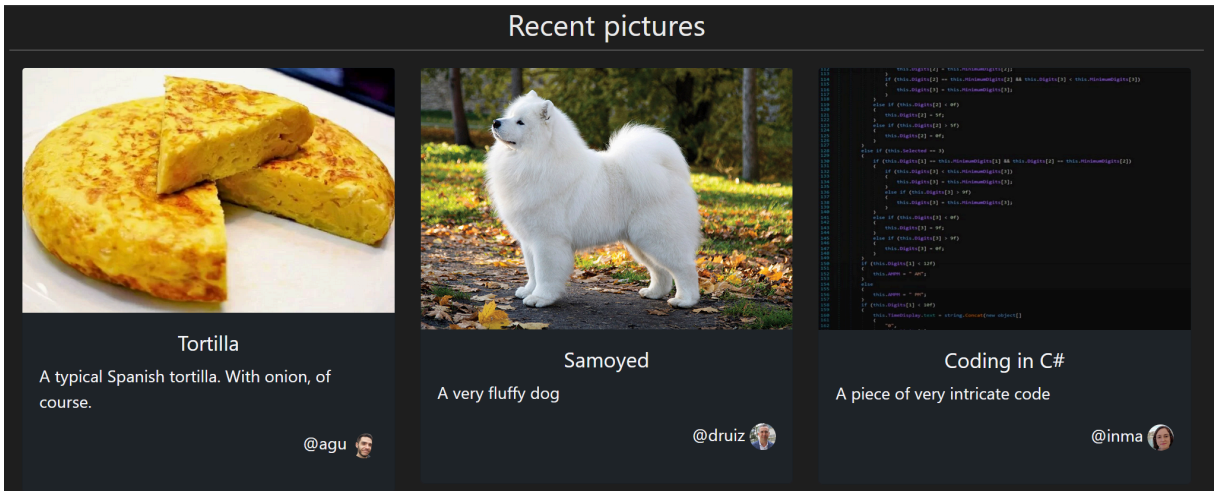
Dado que el renderizador de la galería hace uso, a su vez, del que acabamos de modificar, este cambio hace que automáticamente todas las fotos de la galería muestren el nombre del usuario que las subió:

```
asCard: function (photo) {
    let html = `<div class="col-md-4">
        <div class="card bg-dark text-light">
            <a href="photo_detail.html?photoId=${photo.photoId}">
                <img src="${photo.url}" class="card-img-top">
            </a>

            <div class="card-body">
                <h5 class="card-title text-center">${photo.title}</h5>
                <p class="card-text">${photo.description}</p>
                <p class="text-end">
                    @${photo.username}
                    <img src="${photo.avatarUrl}" class="photo-user-
avatar">
                </p>
            </div>
        </div>
    </div>`;

    let card = parseHTML(html);
    return card;
},
```

Por último, debemos limitar el tamaño de las imágenes de perfil que aparecen junto a las fotos mediante CSS, usando la clase que les hemos asignado:

```
img.photo-user-avatar {
    max-width: 25px;
    height: auto;
    border-radius: 50%;
}
```

El resultado es el siguiente:


Recent pictures
Tortilla
A typical Spanish tortilla. With onion, of course.
@agu
Samoyed
A very fluffy dog
@druiz
Coding in C#
A piece of very intricate code
@inma

8

Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo `commit` a efectuar.
- `git commit -m "mensaje"` crea un nuevo `commit` con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

> **Info**
>
> Versión PDF disponible